Building static websites

It's the </endings> way.



I am seeking to rescue the poor stockinger, the Luddite cropper, the "obsolete" hand-loom weaver, the "utopian" artisan, and even the deluded follower of Joanna Southcott, from the enormous condescension of posterity.

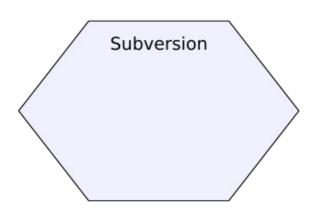
(E.P. Thompson, *The Making of the English Working Class*)



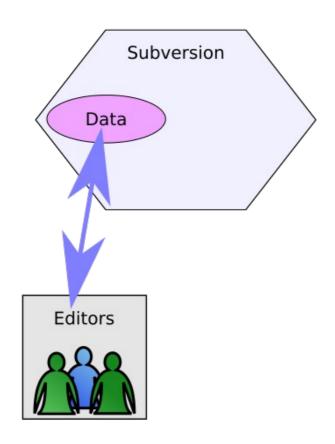
A little detour into history

- Here's an old old site of mine:
 - http://web.uvic.ca/lancenrd/martin/weblang/contents.htm
- It still works. It's also in the Wayback Machine...
- ...where it also still works.
- The HTML still works.
- The JavaScript still works.
- It predates CSS, but if it didn't, the CSS would still work.
- Out of **17** links on the Links page, only **11** work in the Wayback Machine; only **7** work on the current Web.

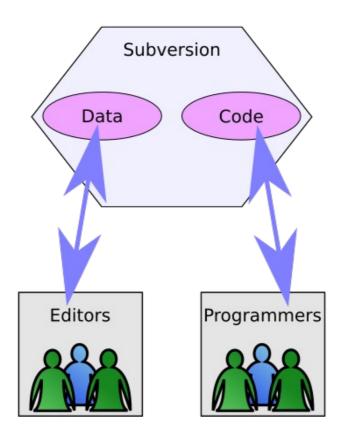
MoEML: The original workflow



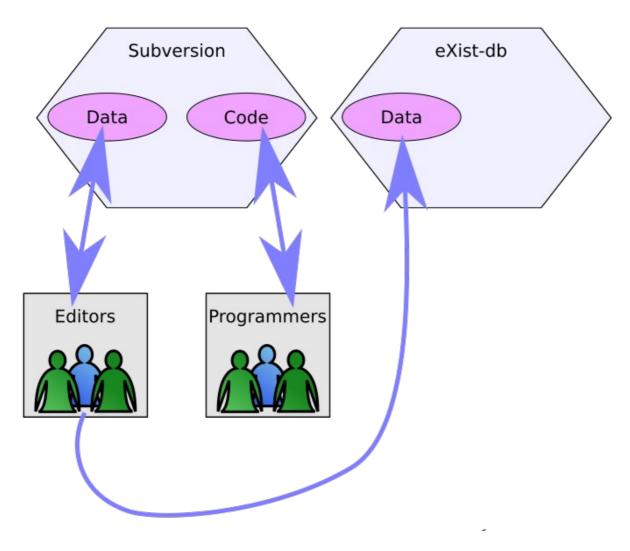




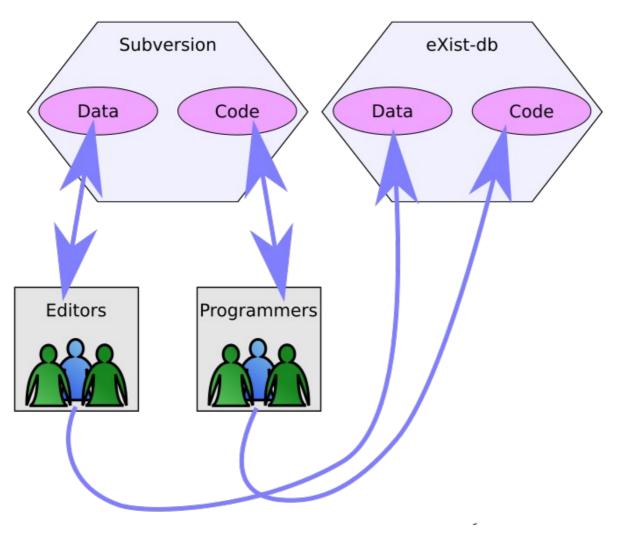




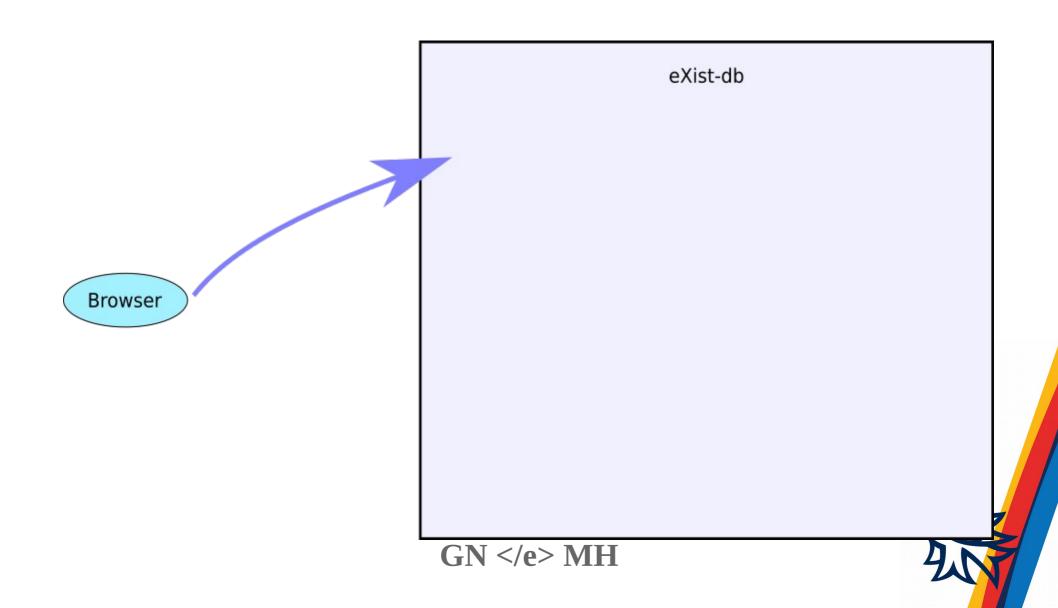


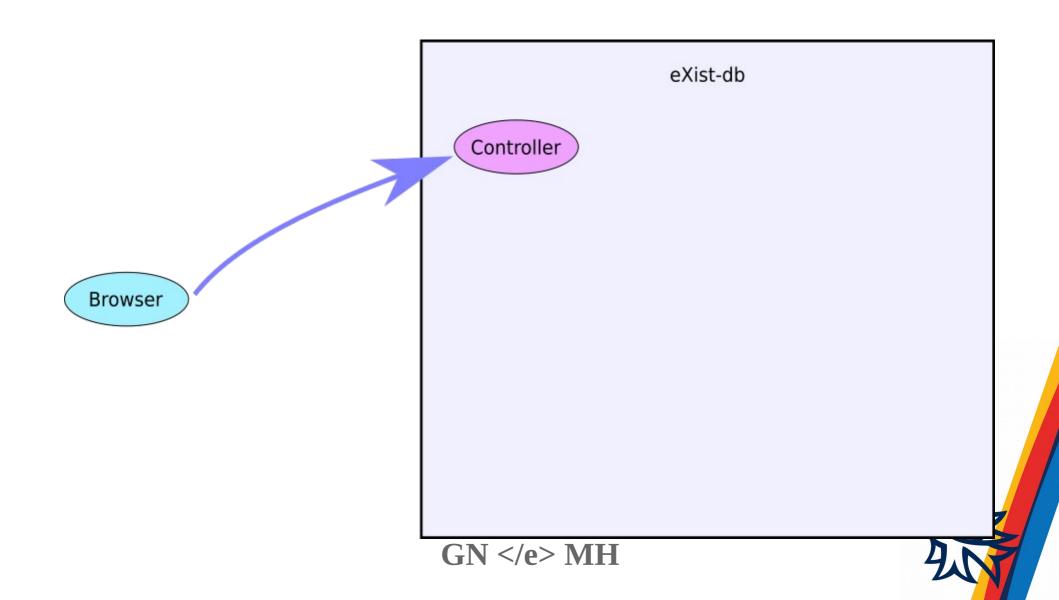


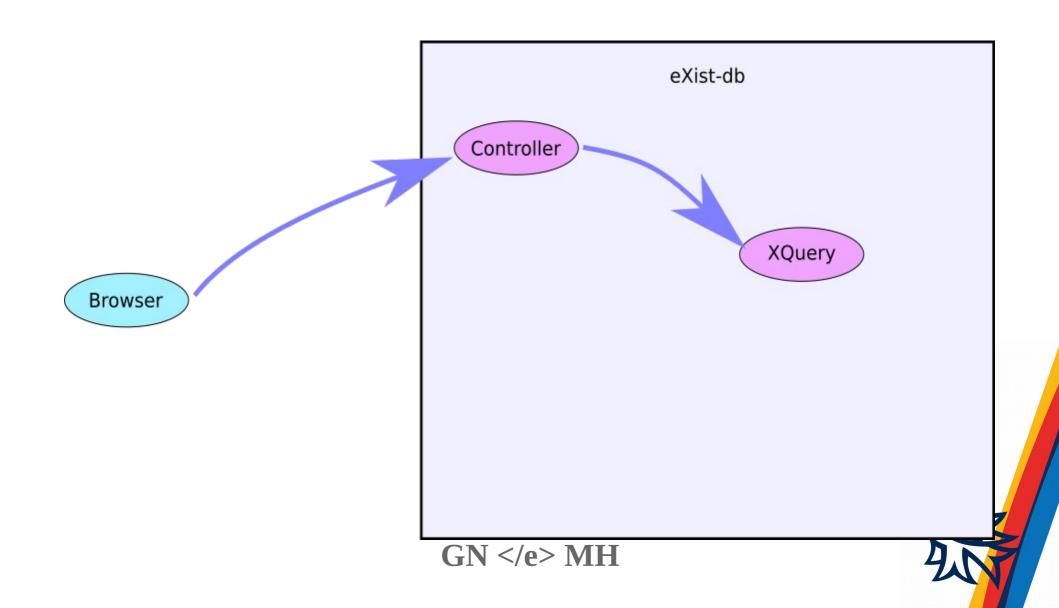


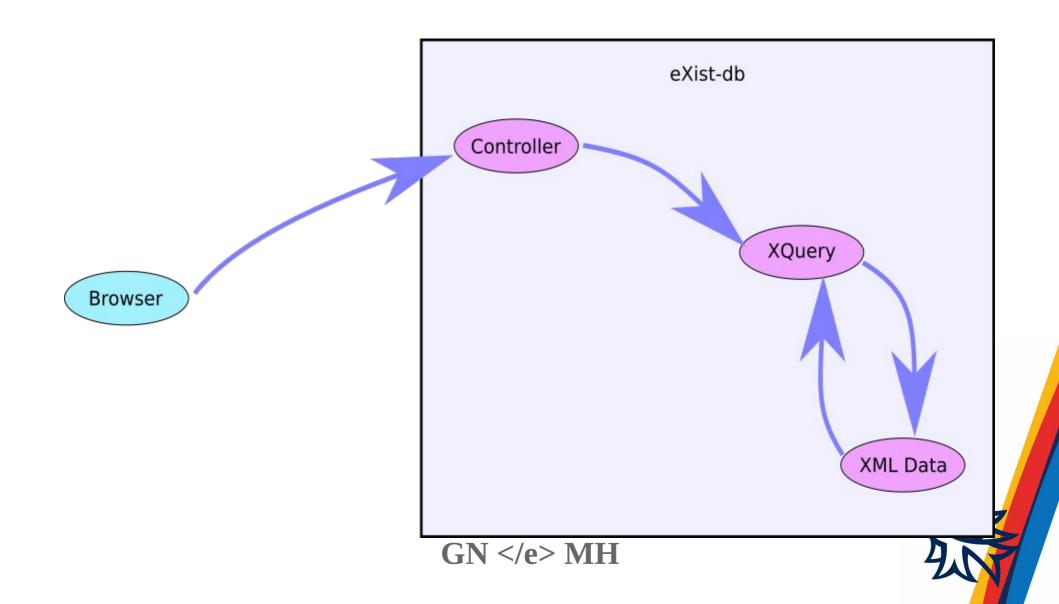


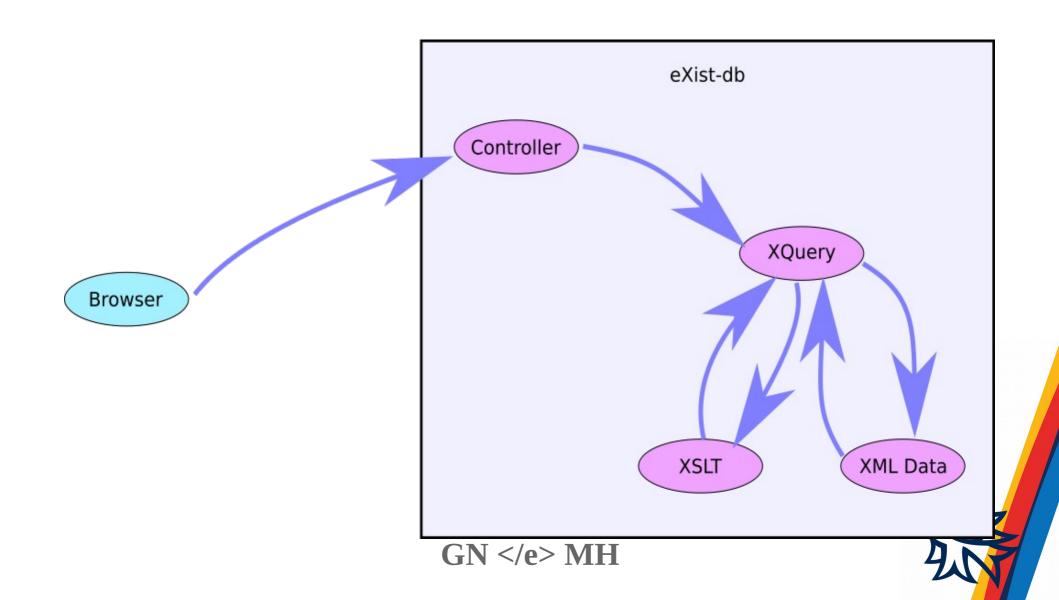


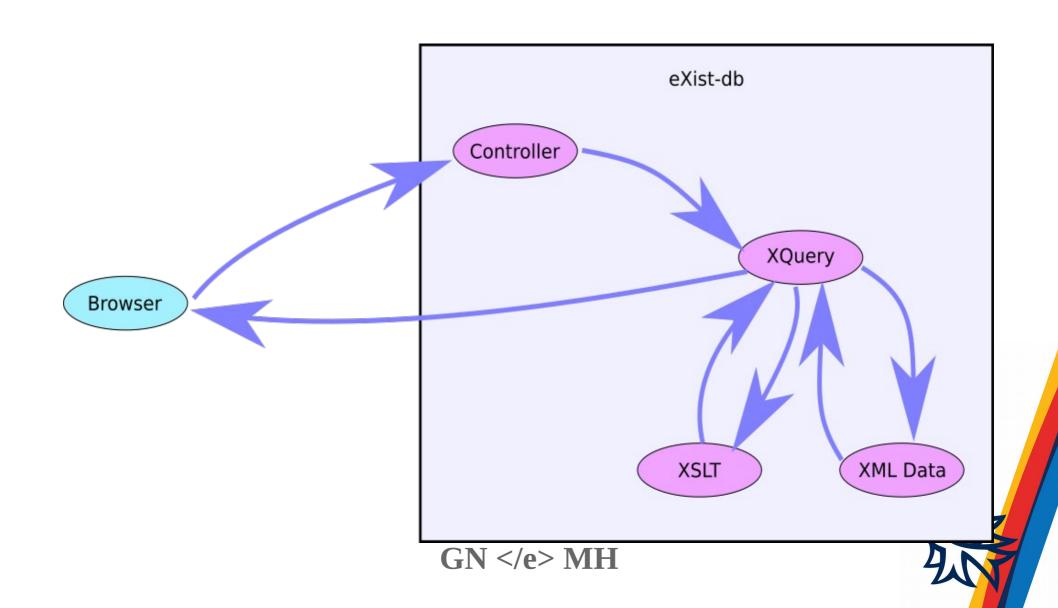


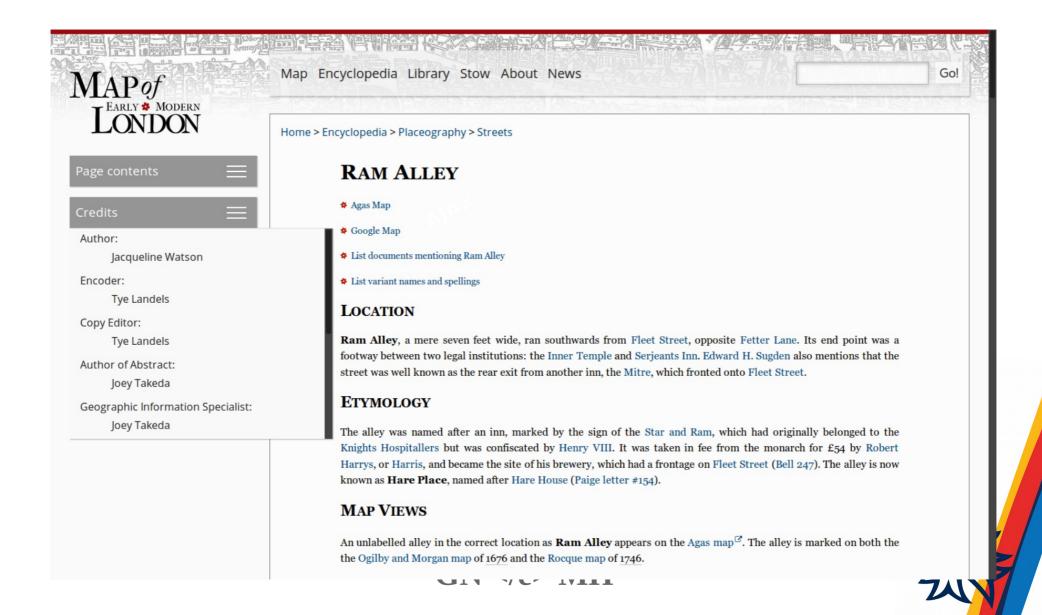


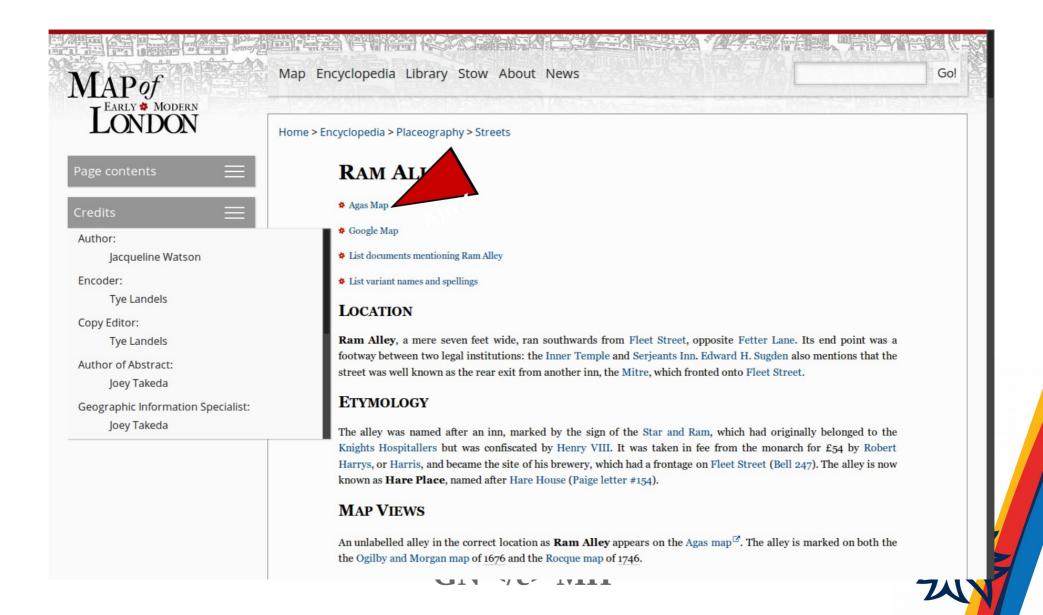


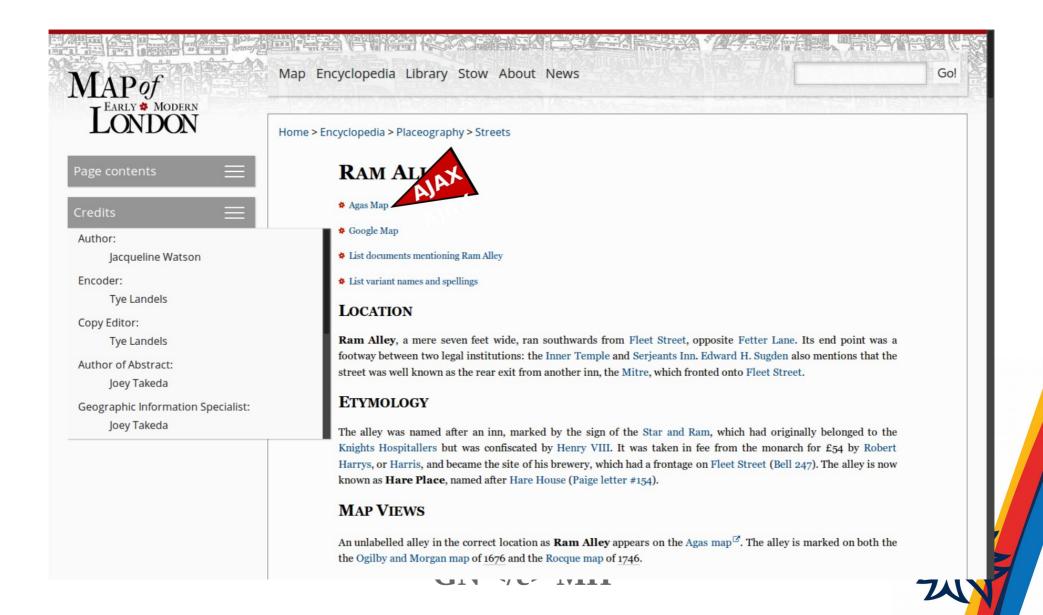


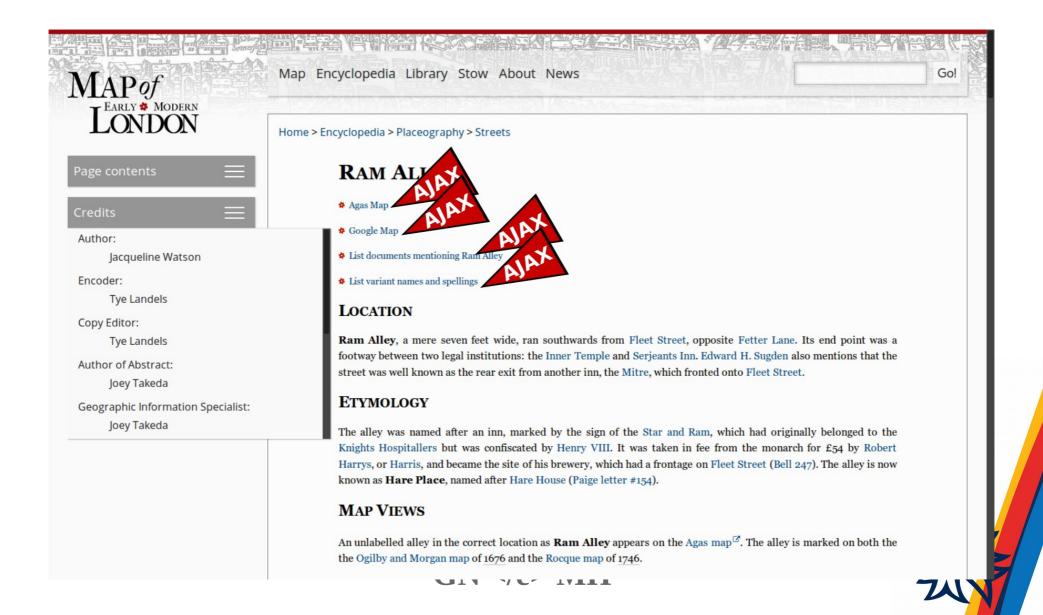


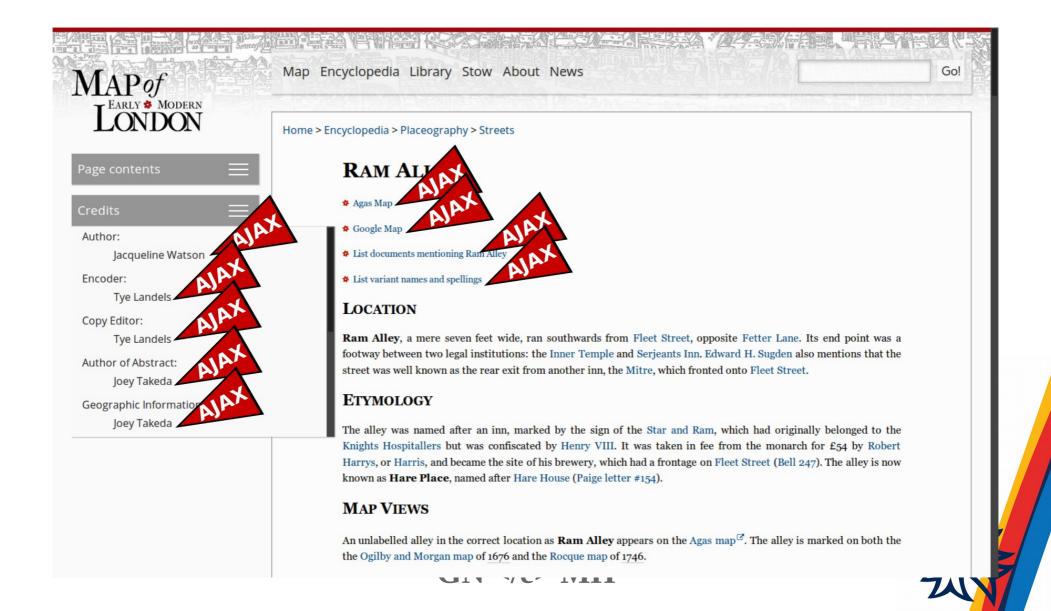


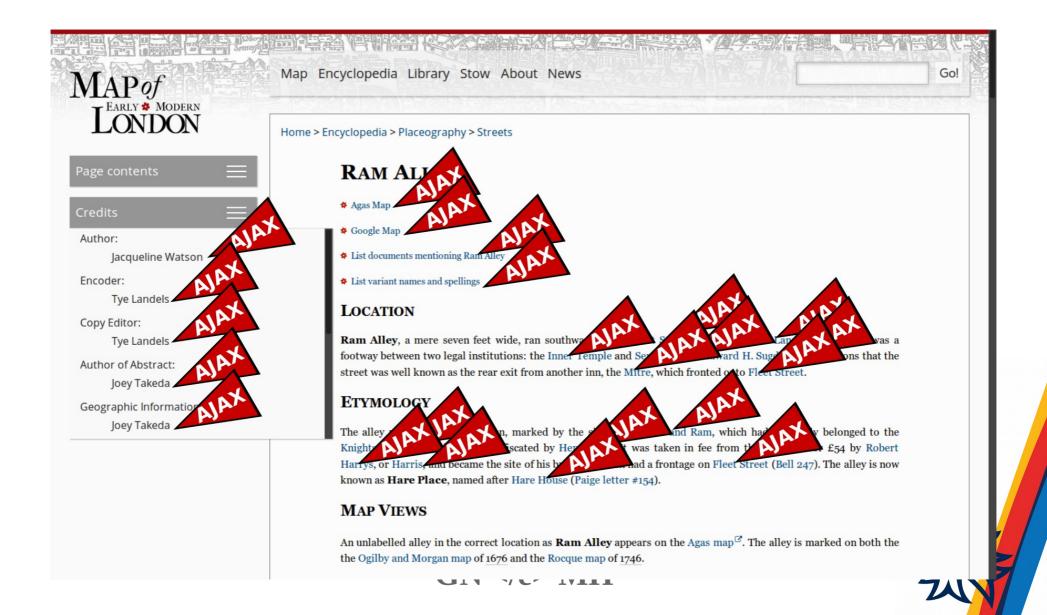


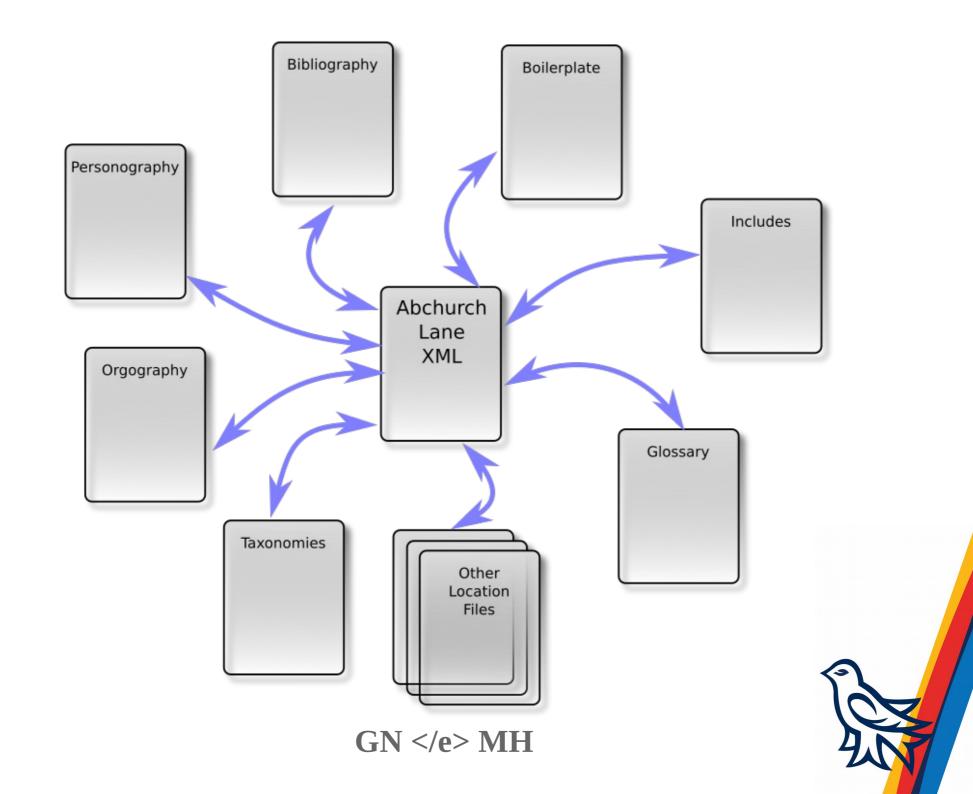












Problems with that model

- Collection-level coherence is difficult to maintain.
- Web applications are difficult to archive.
- Taking and using a single document from the collection is tricky.
- Search has to be specially-crafted for each application.
- Versioning is incoherent and editions don't exist.



Static build principles

- Build everything all the time.
- Validate and diagnose relentlessly.
- Make every document coherent and complete.
- Duplicate everything wherever you need it.
- Make HTML documents degrade gracefully.
- Create every possible version of your documents you can imagine being useful.



The MoEML Build Process (1)

- 1. Validate the **source XML** (RNG and Schematron).
- 2. Create a better version of those documents ("original XML").
- 3. Add *generated XML* versions of all the index, TOC, JSON, tabular and other pages previously created on the fly by eXist.
- 4. Validate this "original XML".
- 5. Create "standalone" versions of all the original XML documents ("standalone XML").
- 6. Validate the standalone XML.
- 7. Create more normative versions of all the XML documents, in which project-specific idiosyncracies are replaced with more conventional encoding strategies ("standard XML").
- 8. Validate the standard XML.



MoEML build process (2)

- 1. Create *TEI Simple* versions of all the docs.
- 9. Validate the TEI Simple documents.
- 10. Create **TEI Lite** versions.
- 9. Validate the TEI Lite.
- 10. Create *KML* output from all the location files.
- 9. Validate the KML.
- 10. Create all the fragments required for responses to *AJAX* requests (12,172 items).
- 11. Create XHTML5 versions of the documents.
- 9. Validate the XHTML5 versions.

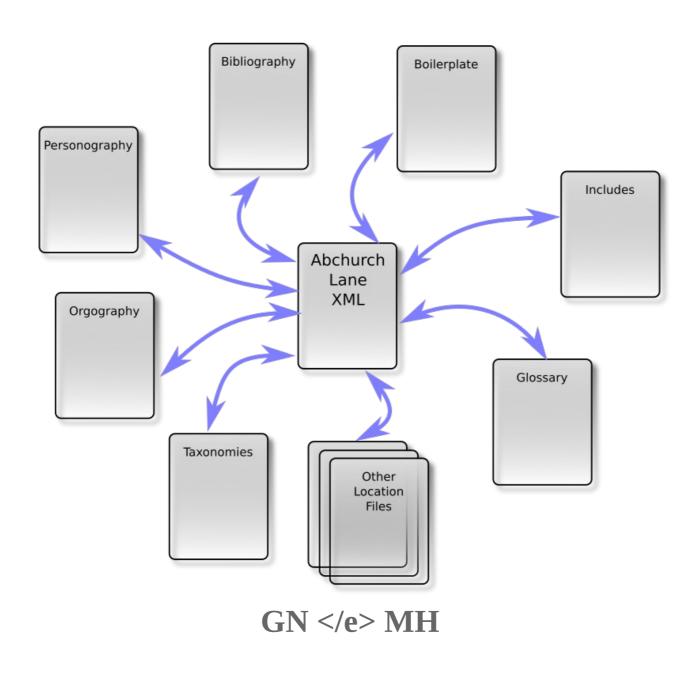


1."**Standalone XML**" contains:

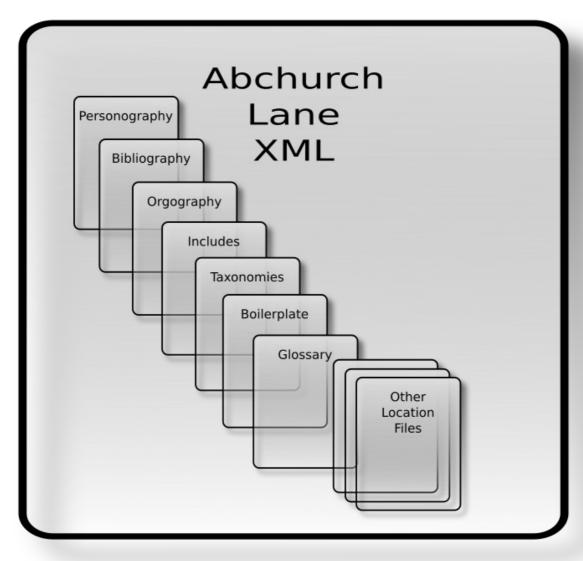
- Every person mentioned
- Every org mentioned
- Every bibliography item referenced
- The abstract for every location mentioned
- All the variant spellings for all locations mentioned
- Full copies of the document type taxonomy and the responsibility taxonomy.



So instead of this:



...we now have this:





GN </e> **MH**

Who does all this building?

Meet Mr Jenkins.





Advantages

- All your documents actually exist.
- All your documents actually contain what they appear to contain.
- Searching your documents will work more intuitively.
- Each build is globally coherent, and can constitute a single edition.
- Every document can function outside the collection.
- Future users can take what they want, in a convenient selfcontained package, in the format they need.
- Everything works without a web server.*

*Search is a special case. But we can do that too...



FUQs and fringe cases

- All applications will benefit from taking a principled approach like this.
- but...
- Some applications won't be as convertible as MoEML.
- Some applications should remain dynamic.



Where is the line?

- Your application is HUGE!
- Your application is COMPLICATED!
- It almost certainly is NOT too big to convert.
- It probably isn't too complicated either.



Features that make this harder

- Your application provides a data service.
- Your application uses a data service.
- Your application's main feature is search.
- Your content is intentionally in constant flux.



Providing/consuming services

- Providing: requires server software.
- Providing: may require a database.
- e.g. Peripleo
- Consuming: maps (tiles/data), social media



Significant searchability

- Your site is premised on the complex searching of large collection(s).
- Potential combinations of search parameters is unknowable.
- Line between document and collection blurry.
- e.g. Canadian Great War Project



Content in constant flux

- Archive: scraper/harvester
- CMS: WordPress, Drupal, Joomla, etc.



Where to next?

- We aren't too sure about the practical limits of this method in some areas (like search).
- Should we use the sophisticated new JavaScript Web APIs?
- We need to try this on a big, database-driven site. Maybe viHistory?



Break free of your CMS

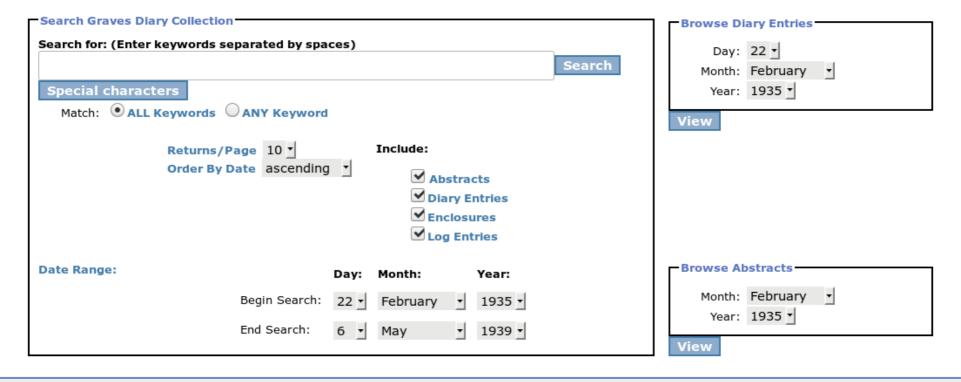
- Convert it to a static publication system like Hugo https://hackernoon.com/wordpress-tostatic-site-generator-hugo-migration-anddeployment-788a69b93e66
- Do it gradually with one of many available plugins: WP2Static, HardyPress, Tome, etc.
- Do it manually with wget.



Appendix: What about search?

Diary of Robert Graves 1935-39 and ancillary material

Copyright St John's College Robert Graves Trust



© 2003 · HCMC · University of Victoria · Site Map · XML Markup · About this Publication



A static search engine



• Standalone all-JavaScript no-backend keyword search with stemming and relevance scoring



How it works (1)

Graves site build process:

```
- ...
```

- TEI XML → XHTML5
- Tokenize HTML body.
- Stem the tokens (Porter stemming).
- For each token, create a JSON file named for the token.
- In the JSON file, place a pointer to each document containing the token, with a score for the number of times it occurs in the document.
- = 11,776 files, 23.2 MB



```
{ "token" : "childish",
 "instances":
  { "docId" : "diary 1938-03-18",
   "docTitle": "Entry for 1938-03-18",
   "docType": "diaryentry",
   "docStartDate": "1938-03-18",
   "docEndDate": "1938-03-18",
   "count": 1 },
  { "docld" : "abstract 1938-03",
   "docTitle": "Abstract for March 1938",
   "docType": "abstract",
   "docStartDate": "1938-03-01",
   "docEndDate": "1938-03-31",
   "count": 1 }
```



How it works (2)

Search page:

- User types in keywords.
- Keywords are stemmed by JavaScript.
- For each unique token, retrieve the JSON file named for it.
- Combine the scores for each document across the tokens.
- Do additional filtering by date and document type.
- Present results ordered by score.



Search Graves Diary Collect Search for: (Enter keyword: For proper nam							
love Search							
Include:							
✓ Abstracts ✓ Diary Entries ✓ Enclosures ✓ Log Entries							
Date Range:		Day:	Month:		Year:		
	Begin Search:	1 🔻	January	•	1937 🛨		
	End Search:	6	May	•	1939 🛨		

Searched for: love Documents found: 48

- Enclosure Letter to RG and LR from Karl Goldschmidt 1938-10-17 (Score: 6)
- Enclosure 5-page letter to RG from Jenny in Liverpool 1938-12-12 (Score: 5)
- Enclosure Letter to LR from Margaret Russell 1938-09-03 (Score: 4)
- Enclosure Letter to RG from Ros Graves 1939-01-12 (Score: 3)
- Enclosure Postcard to RG from David Graves 1938-03-31 (Score: 3)
- Enclosure Letter from Catherine Nicholson, signed Kate 1937-11-30 (Score: 2)
- Enclosure Letter from David Graves 1938-10-01 (Score: 2)
- Enclosure Letter to RG from Jenny Nicholson 1937-08-05 (Score: 2)
- Entry for 1939-03-07 (Score: 2)
- Enclosure Letter to RG and LR from Sam Graves 1938-11-04 (Score: 2)
- Entry for 1938-04-08 (Score: 2)



Pros and cons

- Lightning fast
- Works anywhere
- Usable and effective
- No keywords-in-context X
- Only practical for small projects X

